CHAPTER 6

Advanced Routing

Chapter 5 dealt with the IPv4 routing subsystem. This chapter continues with the routing subsystem and discusses advanced IPv4 routing topics such as Multicast Routing, Multipath Routing, Policy Routing, and more. This book deals with the Linux Kernel Networking implementation—it does not delve into the internals of userspace Multicast Routing daemons implementation, which are quite complex and beyond the scope of the book. I do, however, discuss to some extent the interaction between a userspace multicast routing daemon and the multicast layer in the kernel. I also briefly discuss the Internet Group Management Protocol (IGMP) protocol, which is the basis of multicast group membership management; adding and deleting multicast group members is done by the IGMP protocol. Some basic knowledge of IGMP is needed to understand the interaction between a multicast host and a multicast router.

Multipath Routing is the ability to add more than one nexthop to a route. Policy Routing enables configuring routing policies that are not based solely on the destination address. I start with describing Multicast Routing.

Multicast Routing

Chapter 4 briefly mentions Multicast Routing, in the "Receiving IPv4 Multicast Packets" section. I will now discuss it in more depth. Sending multicast traffic means sending the same packet to multiple recipients. This feature can be useful in streaming media, audio/video conferencing, and more. It has a clear advantage over unicast traffic in terms of saving network bandwidth. Multicast addresses are defined as Class D addresses. The Classless Inter-Domain Routing (CIDR) prefix of this group is 224.0.0.0/4. The range of IPv4 multicast addresses is from 224.0.0.0 to 239.255.255.255. Handling Multicast Routing must be done in conjunction with a userspace routing daemon which interacts with the kernel. According to the Linux implementation, Multicast Routing cannot be handled solely by the kernel code without this userspace Routing daemon, as opposed to Unicast Routing. There are various multicast daemons: for example: `mrouted`, which is based on an implementation of the Distance Vector Multicast Routing Protocol (DVMRP), or `pimd`, which is based on the Protocol-Independent Multicast protocol (PIM). The DVMRP protocol is defined in RFC 1075, and it was the first multicast routing protocol. It is based on the Routing Information Protocol (RIP) protocol.

The PIM protocol has two versions, and the kernel supports both of them (CONFIG_IP_PIMSM_V1 and CONFIG_IP_PIMSM_V2). PIM has four different modes: PIM-SM (PIM Sparse Mode), PIM-DM (PIM Dense Mode), PIM Source-Specific Multicast (PIM-SSM) and Bidirectional PIM. The protocol is called *protocol-independent* because it is not dependent on any particular routing protocol for topology discovery. This section discusses the interaction between the userspace daemon and the kernel multicast routing layer. Delving into the internals of the PIM protocol or the DVMRP protocol (or any other Multicast Routing protocol) is beyond the scope of this book. Normally, the Multicast Routing lookup is based on the source and destination addresses. There is a "Multicast Policy Routing" kernel feature, which is the parallel to the unicast policy routing kernel feature that was mentioned in Chapter 5 and which is also discussed in the course of this chapter. The multicast policy routing protocol is implemented using the Policy Routing API (for example, it calls the `fib_rules_lookup()` method to perform a lookup, creates a `fib_rules_ops` object, and registers it with the `fib_rules_register()` method, and so on). With Multicast Policy Routing, the routing can be based on additional criteria, like the ingress network interfaces. Moreover, you can work with more than one multicast routing table. In order to work with Multicast Policy Routing, IP_MROUTE_MULTIPLE_TABLES must be set.

Figure 6-1 shows a simple IPv4 Multicast Routing setup. The topology is very simple: the laptop, on the left, joins a multicast group (224.225.0.1) by sending an IGMP packet (IP_ADD_MEMBERSHIP). The IGMP protocol is discussed in the next section, “The IGMP Protocol.” The AMD server, in the middle, is configured as a multicast router, and a userspace multicast routing daemon (like `pimd` or `mrouted`) runs on it. The Windows server, on the right, which has an IP address of 192.168.2.10, sends multicast traffic to 224.225.0.1; this traffic is forwarded to the laptop via the multicast router. Note that the Windows server itself did not join the 224.225.0.1 multicast group. Running `ip route add 224.0.0.0/4 dev <networkDeviceName>` tells the kernel to send all multicast traffic via the specified network device.

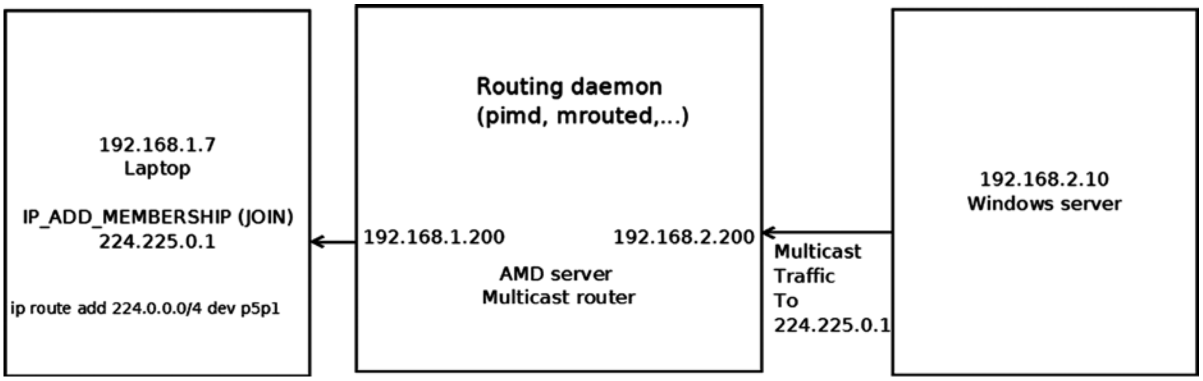

Figure 6-1. *Simple Multicast Routing setup*

The next section discusses the IGMP protocol, which is used for the management of multicast group membership.

The IGMP Protocol

The IGMP protocol is an integral part of IPv4 multicast. It must be implemented on each node that supports IPv4 multicast. In IPv6, multicast management is handled by the MLD (Multicast Listener Discovery) protocol, which uses ICMPv6 messages, discussed in Chapter 8. With the IGMP protocol, multicast group memberships are established and managed. There are three versions of IGMP:

1. *IGMPv1 (RFC 1112):* Has two types of messages—host membership report and host membership query. When a host wants to join a multicast group, it sends a membership report message. Multicast routers send membership queries to discover which host multicast groups have members on their attached local networks. Queries are addressed to the all-hosts group address (224.0.0.1, IGMP_ALL_HOSTS) and carry a TTL of 1 so that the membership query will not travel outside of the LAN.

2. *IGMPv2 (RFC 2236):* This is an extension of IGMPv1. The IGMPv2 protocol adds three new messages:

    a. Membership Query (0x11): There are two sub-types of Membership Query messages: General Query, used to learn which groups have members on an attached network, and Group-Specific Query, used to learn whether a particular group has any members on an attached network.

    b. Version 2 Membership Report (0x16).

    c. Leave Group (0x17).

■ **Note** IGMPv2 also supports Version 1 Membership Report message, for backward compatibility with IGMPv1. See RFC 2236, section 2.1.

---

3. *IGMPv3 (RFC 3376, updated by RFC 4604):* This major revision of the protocol adds a feature called source filtering. This means that when a host joins a multicast group, it can specify a set of source addresses from which it will receive multicast traffic. The source filters can also exclude source addresses. To support the source filtering feature, the socket API was extended; see RFC 3678, "Socket Interface Extensions for Multicast Source Filters." I should also mention that the multicast router periodically (about every two minutes) sends a membership query to 224.0.0.1, the all-hosts multicast group address. A host that receives a membership query responds with a membership report. This is implemented in the kernel by the `igmp_rcv()` method: getting an IGMP_HOST_MEMBERSHIP_QUERY message is handled by the `igmp_heard_query()` method.

■ **Note** The kernel implementation of IPv4 IGMP is in `net/core/igmp.c`, `include/linux/igmp.h` and `include/uapi/linux/igmp.h`.

The next section examines the fundamental data structure of IPv4 Multicast Routing, the multicast routing table, and its Linux implementation.

The Multicast Routing Table

The multicast routing table is represented by a structure named `mr_table`. Let's take a look at it:

```
struct mr_table {
    struct list_head     list;
#ifdef CONFIG_NET_NS
    struct net           *net;
#endif
    u32                  id;
    struct sock __rcu    *mroute_sk;
    struct timer_list    ipmr_expire_timer;
    struct list_head     mfc_unres_queue;
    struct list_head     mfc_cache_array[MFC_LINES];
    struct vif_device    vif_table[MAXVIFS];
    . . .
};
```

(net/ipv4/ipmr.c)

The following is a description of some members of the mr_table structure:

- net: The network namespace associated with the multicast routing table; by default it is the initial network namespace, init_net. Network namespaces are discussed in Chapter 14.
- id: The multicast routing table id; it is RT_TABLE_DEFAULT (253) when working with a single table.
- mroute_sk: This pointer represents a reference to the userspace socket that the kernel keeps. The mroute_sk pointer is initialized by calling setsockopt() from the userspace with the MRT_INIT socket option and is nullified by calling setsockopt() with the MRT_DONE socket option. The interaction between the userspace and the kernel is based on calling the setsockopt() method, on sending IOCTLs from userspace, and on building IGMP packets and passing them to the Multicast Routing daemon by calling the sock_queue_rcv_skb() method from the kernel.
- ipmr_expire_timer: Timer of cleaning unresolved multicast routing entries. This timer is initialized when creating a multicast routing table, in the ipmr_new_table() method, and removed when removing a multicast routing table, by the ipmr_free_table() method.
- mfc_unres_queue: A queue of unresolved routing entries.
- mfc_cache_array: A cache of the routing entries, with 64 (MFC_LINES) entries, discussed shortly in the next section.
- vif_table[MAXVIFS]: An array of 32 (MAXVIFS) vif_device objects. Entries are added by the vif_add() method and deleted by the vif_delete() method. The vif_device structure represents a virtual multicast routing network interface; it can be based on a physical device or on an IPIP (IP over IP) tunnel. The vif_device structure is discussed later in "The Vif Device" section.

I have covered the multicast routing table and mentioned its important members, such as the Multicast Forwarding Cache (MFC) and the queue of unresolved routing entries. Next I will look at the MFC, which is embedded in the multicast routing table object and plays an important role in Multicast Routing.

The Multicast Forwarding Cache (MFC)

The most important data structure in the multicast routing table is the MFC, which is in fact an array of cache entries (mfc_cache objects). This array, named mfc_cache_array, is embedded in the multicast routing table (mr_table) object. It has 64 (MFC_LINES) elements. The index of this array is the hash value (the hash function takes two parameters—the multicast group address and the source IP address; see the description of the MFC_HASH macro in the "Quick Reference" section at the end of this chapter).

Usually there is only one multicast routing table, which is an instance of the mr_table structure, and a reference to it is kept in the IPv4 network namespace (net->ipv4.mrt). The table is created by the ipmr_rules_init() method, which also assigns net->ipv4.mrt to point to the multicast routing table that was created. When working with the multicast policy routing feature mentioned earlier, there can be multiple multicast policy routing tables. In both cases, you get the routing table using the same method, ipmr_fib_lookup(). The ipmr_fib_lookup() method gets three parameters as an input: the network namespace, the flow, and a pointer to the mr_table object which it should fill. Normally, it simply sets the specified `mr_table` pointer to be `net->ipv4.mrt`; when working with multiple tables (IP_MROUTE_MULTIPLE_TABLES is set), the implementation is more complex. Let's take a look at the `mfc_cache` structure:

```
struct mfc_cache {
    struct list_head list;
    __be32 mfc_mcastgrp;
    __be32 mfc_origin;
    vifi_t mfc_parent;
    int mfc_flags;
    union {
            struct {
                    unsigned long expires;
                    struct sk_buff_head unresolved; /* Unresolved buffers */
            } unres;
            struct {
                    unsigned long last_assert;
                    int minvif;
                    int maxvif;
                    unsigned long bytes;
                    unsigned long pkt;
                    unsigned long wrong_if;
                    unsigned char ttls[MAXVIFS];    /* TTL thresholds */
            } res;
    } mfc_un;
    struct rcu_head rcu;
 };
```

(`include/linux/mroute.h`)

The following is a description of some members of the `mfc_cache` structure:

- `mfc_mcastgrp`: the address of the multicast group that the entry belongs to.
- `mfc_origin`: The source address of the route.
- `mfc_parent`: The source interface.
- `mfc_flags`: The flags of the entry. Can have one of these values:
    - MFC_STATIC: When the route was added statically and not by a multicast routing daemon.
    - MFC_NOTIFY: When the RTM_F_NOTIFY flag of the routing entry was set. See the `rt_fill_info()` method and the `ipmr_get_route()` method for more details.
- The `mfc_un` union consists of two elements:
    - `unres`: Unresolved cache entries.
    - `res`: Resolved cache entries.

The first time an SKB of a certain flow reaches the kernel, it is added to the queue of unresolved entries (mfc_un.unres.unresolved), where up to three SKBs can be saved. If there are three SKBs in the queue, the packet is not appended to the queue but is freed, and the ipmr_cache_unresolved() method returns -ENOBUFS ("No buffer space available"):

```
static int ipmr_cache_unresolved(struct mr_table *mrt, vifi_t vifi, struct sk_buff *skb)
{
         . . .
         if (c->mfc_un.unres.unresolved.qlen > 3) {
                 kfree_skb(skb);
                 err = -ENOBUFS;
        } else {
            . . .

}
```

(net/ipv4/ipmr.c)

This section described the MFC and its important members, including the queue of resolved entries and the queue of unresolved entries. The next section briefly describes what a multicast router is and how it is configured in Linux.

Multicast Router

In order to configure a machine as a multicast router, you should set the CONFIG_IP_MROUTE kernel configuration option. You should also run some routing daemon such as pimd or mrouted, as mentioned earlier. These routing daemons create a socket to communicate with the kernel. In pimd, for example, you create a raw IGMP socket by calling socket(AF_INET, SOCK_RAW, IPPROTO_IGMP). Calling setsockopt() on this socket triggers sending commands to the kernel, which are handled by the ip_mroute_setsockopt() method. When calling setsockopt() on this socket from the routing daemon with MRT_INIT, the kernel is set to keep a reference to the userspace socket in the mroute_sk field of the mr_table object that is used, and the mc_forwarding procfs entry (/proc/sys/net/ipv4/conf/all/mc_forwarding) is set by calling IPV4_DEVCONF_ALL(net, MC_FORWARDING)++. Note that the mc_forwarding procfs entry is a read-only entry and can't be set from userspace. You can't create another instance of a multicast routing daemon: when handling the MRT_INIT option, the ip_mroute_setsockopt() method checks whether the mroute_sk field of the mr_table object is initialized and returns -EADDRINUSE if so. Adding a network interface is done by calling setsockopt() on this socket with MRT_ADD_VIF, and deleting a network interface is done by calling setsockopt() on this socket with MRT_DEL_VIF. You can pass the parameters of the network interface to these setsockopt() calls by passing a vifctl object as the optval parameter of the setsockopt() system call. Let's take a look at the vifctl structure:

```
struct vifctl {
    vifi_t    vifc_vifi;                /* Index of VIF */
    unsigned char vifc_flags;            /* VIFF_ flags */
    unsigned char vifc_threshold;        /* ttl limit */
    unsigned int vifc_rate_limit;        /* Rate limiter values (NI) */
    union {
        struct in_addr vifc_lcl_addr;     /* Local interface address */
        int            vifc_lcl_ifindex;  /* Local interface index   */
    };
    struct in_addr vifc_rmt_addr;    /* IPIP tunnel addr */
};
```

(include/uapi/linux/mroute.h)

The following is a description of some members of the vifctl structure:

- vifc_flags can be:
  - VIFF_TUNNEL: When you want to use an IPIP tunnel.
  - VIFF_REGISTER: When you want to register the interface.
  - VIFF_USE_IFINDEX: When you want to use the local interface index and not the local interface IP address; in such a case, you will set the vifc_lcl_ifindex to be the local interface index. The VIFF_USE_IFINDEX flag is available for 2.6.33 kernel and above.
- vifc_lcl_addr: The local interface IP address. (This is the default—no flag should be set for using it).
- vifc_lcl_ifindex: The local interface index. It should be set when the VIFF_USE_IFINDEX flag is set in vifc_flags.
- vifc_rmt_addr: The address of the remote node of a tunnel.

When the multicast routing daemon is closed, the setsockopt() method is called with an MRT_DONE option. This triggers calling the mrtsock_destruct() method to nullify the mroute_sk field of the mr_table object that is used and to perform various cleanups.

This section covered what a multicast router is and how it is configured in Linux. I also examined the vifctl structure. Next, I look at the Vif device, which represents a multicast network interface.

The Vif Device

Multicast Routing supports two modes: direct multicast and multicast encapsulated in a unicast packet over a tunnel. In both cases, the same object is used (an instance of the vif_device structure) to represent the network interface. When working over a tunnel, the VIFF_TUNNEL flag will be set. Adding and deleting a multicast interface is done by the vif_add() method and by the vif_delete() method, respectively. The vif_add() method also sets the device to support multicast by calling the dev_set_allmulti(dev, 1) method, which increments the allmulti counter of the specified network device (net_device object). The vif_delete() method calls dev_set_allmulti(dev, -1) to decrement the allmulti counter of the specified network device (net_device object). For more details about the dev_set_allmulti() method, see appendix A. Let's take a look at the vif_device structure; its members are quite self-explanatory:

```
struct vif_device {
        struct net_device       *dev;       /* Device we are using */
        unsigned long   bytes_in,bytes_out;
        unsigned long   pkt_in,pkt_out;     /* Statistics                  */
        unsigned long   rate_limit;         /* Traffic shaping (NI)        */
        unsigned char   threshold;          /* TTL threshold               */
        unsigned short  flags;              /* Control flags               */
        __be32          local,remote;       /* Addresses(remote for tunnels)*/
        int             link;               /* Physical interface index    */
};
```

(include/linux/mroute.h)

In order to receive multicast traffic, a host must join a multicast group. This is done by creating a socket in userspace and calling setsockopt() with IPPROTO_IP and with the IP_ADD_MEMBERSHIP socket option. The userspace application also creates an ip_mreq object where it initializes the request parameters, like the desired group multicast address and the source IP address of the host (see the netinet/in.h userspace header). The setsockopt() call is handled in the kernel by the ip_mc_join_group() method, in net/ipv4/igmp.c. Eventually, the multicast address is added by the ip_mc_join_group() method to a list of multicast addresses (mc_list), which is a member of the in_device object. A host can leave a multicast group by calling setsockopt() with IPPROTO_IP and with the IP_DROP_MEMBERSHIP socket option. This is handled in the kernel by the ip_mc_leave_group() method, in net/ipv4/igmp.c. A single socket can join up to 20 multicast groups (sysctl_igmp_max_memberships). Trying to join more than 20 multicast groups by the same socket will fail with the -ENOBUFS error ("No buffer space available.") See the ip_mc_join_group() method implementation in net/ipv4/igmp.c.

IPv4 Multicast Rx Path

Chapter 4's "Receiving IPv4 Multicast Packets" section briefly discusses how multicast packets are handled. I will now describe this in more depth. My discussion assumes that our machine is configured as a multicast router; this means, as was mentioned earlier, that CONFIG_IP_MROUTE is set and a routing daemon like pimd or mrouted runs on this host. Multicast packets are handled by the ip_route_input_mc() method, in which a routing table entry (an rtable object) is allocated and initialized, and in which the input callback of the dst object is set to be ip_mr_input(), in case CONFIG_IP_MROUTE is set. Let's take a look at the ip_mr_input() method:

```
int ip_mr_input(struct sk_buff *skb)
{
        struct mfc_cache *cache;
        struct net *net = dev_net(skb->dev);
```

First the local flag is set to true if the packet is intended for local delivery, as the ip_mr_input() method also handles local multicast packets.

```
int local = skb_rtable(skb)->rt_flags & RTCF_LOCAL;
struct mr_table *mrt;

/* Packet is looped back after forward, it should not be
* forwarded second time, but still can be delivered locally.
*/
if (IPCB(skb)->flags & IPSKB_FORWARDED)
        goto dont_forward;
```

Normally, when working with a single multicast routing table, the ipmr_rt_fib_lookup() method simply returns the net->ipv4.mrt object:

```
mrt = ipmr_rt_fib_lookup(net, skb);
if (IS_ERR(mrt)) {
        kfree_skb(skb);
        return PTR_ERR(mrt);
}
if (!local) {
```

IGMPv3 and some IGMPv2 implementations set the router alert option (IPOPT_RA) in the IPv4 header when sending JOIN or LEAVE packets. See the igmpv3_newpack() method in net/ipv4/igmp.c:

```
if (IPCB(skb)->opt.router_alert) {
```

The ip_call_ra_chain() method (net/ipv4/ip_input.c) calls the raw_rcv() method to pass the packet to the userspace raw socket, which listens. The ip_ra_chain object contains a reference to the multicast routing socket, which is passed as a parameter to the raw_rcv() method. For more details, look at the ip_call_ra_chain() method implementation, in net/ipv4/ip_input.c:

```
if (ip_call_ra_chain(skb))
        return 0;
```

There are implementations where the router alert option is not set, as explained in the following comment; these cases must be handled as well, by calling the raw_rcv() method directly:

```
} else if (ip_hdr(skb)->protocol == IPPROTO_IGMP) {
        /* IGMPv1 (and broken IGMPv2 implementations sort of
        * Cisco IOS <= 11.2(8)) do not put router alert
        * option to IGMP packets destined to routable
        * groups. It is very bad, because it means
        * that we can forward NO IGMP messages.
        */
        struct sock *mroute_sk;
```

The mrt->mroute_sk socket is a copy in the kernel of the socket that the multicast routing userspace application created:

```
mroute_sk = rcu_dereference(mrt->mroute_sk);
        if (mroute_sk) {
        nf_reset(skb);
        raw_rcv(mroute_sk, skb);
        return 0;
        }
     }
}
```

First a lookup in the multicast routing cache, mfc_cache_array, is performed by calling the ipmr_cache_find() method. The hash key is the destination multicast group address and the source IP address of the packet, taken from the IPv4 header:

```
cache = ipmr_cache_find(mrt, ip_hdr(skb)->saddr, ip_hdr(skb)->daddr);
if (cache == NULL) {
```

A lookup in the virtual devices array (vif_table) is performed to see whether there is a corresponding entry which matches the incoming network device (skb->dev):

```
int vif = ipmr_find_vif(mrt, skb->dev);
```

The ipmr_cache_find_any() method handles the advanced feature of multicast proxy support (which is not discussed in this book):

```
        if (vif >= 0)
                cache = ipmr_cache_find_any(mrt, ip_hdr(skb)->daddr,
                                        vif);
}
```

```
/*
*      No usable cache entry
*/
if (cache == NULL) {
        int vif;
```

If the packet is destined to the local host, deliver it:

```
if (local) {
        struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
        ip_local_deliver(skb);
        if (skb2 == NULL)
                return -ENOBUFS;
        skb = skb2;
}

read_lock(&mrt_lock);
vif = ipmr_find_vif(mrt, skb->dev);
if (vif >= 0) {
```

The `ipmr_cache_unresolved()` method creates a multicast routing entry (`mfc_cache` object) by calling the `ipmr_cache_alloc_unres()` method. This method creates a cache entry (`mfc_cache` object) and initializes its expiration time interval (by setting `mfc_un.unres.expires`). Let's take a look at this very short method, `ipmr_cache_alloc_unres()`:

```
static struct mfc_cache *ipmr_cache_alloc_unres(void)
{
    struct mfc_cache *c = kmem_cache_zalloc(mrt_cachep, GFP_ATOMIC);

    if (c) {
        skb_queue_head_init(&c->mfc_un.unres.unresolved);
```

Setting the expiration time interval:

```
        c->mfc_un.unres.expires = jiffies + 10*HZ;
    }
    return c;
}
```

If the routing daemon does not resolve the routing entry within its expiration interval, the entry is removed from the queue of the unresolved entries. When creating a multicast routing table (by the `ipmr_new_table()` method), its timer (`ipmr_expire_timer`) is set. This timer invokes the `ipmr_expire_process()` method periodically. The `ipmr_expire_process()` method iterates over all the unresolved cache entries in the queue of unresolved entries (`mfc_unres_queue` of the `mrtable` object) and removes the expired unresolved cache entries.

After creating the unresolved cache entry, the `ipmr_cache_unresolved()` method adds it to the queue of unresolved entries (`mfc_unres_queue` of the multicast table, `mrtable`) and increments by 1 the unresolved queue length (`cache_resolve_queue_len` of the multicast table, `mrtable`). It also calls the `ipmr_cache_report()` method, which builds an IGMP message (IGMPMSG_NOCACHE) and delivers it to the userspace multicast routing daemon by calling eventually the `sock_queue_rcv_skb()` method.

I mentioned that the userspace routing daemon should resolve the routing within some time interval. I will not delve into how this is implemented in userspace. Note, however, that once the routing daemon decides it should resolve an unresolved entry, it builds the cache entry parameters (in an mfcctl object) and calls setsockopt() with the MRT_ADD_MFC socket option, then it passes the mfcctl object embedded in the optval parameter of the setsockopt() system call; this is handled in the kernel by the ipmr_mfc_add() method:

```
                int err2 = ipmr_cache_unresolved(mrt, vif, skb);
                read_unlock(&mrt_lock);

                return err2;
        }
        read_unlock(&mrt_lock);
        kfree_skb(skb);
        return -ENODEV;
}

read_lock(&mrt_lock);
```

If a cache entry was found in the MFC, call the ip_mr_forward() method to continue the packet traversal:

```
        ip_mr_forward(net, mrt, skb, cache, local);
        read_unlock(&mrt_lock);

        if (local)
                return ip_local_deliver(skb);

        return 0;

dont_forward:
        if (local)
                return ip_local_deliver(skb);
        kfree_skb(skb);
        return 0;
}
```

This section detailed the IPv4 Multicast Rx path and the interaction with the routing daemon in this path. The next section describes the multicast routing forwarding method, ip_mr_forward().

The ip_mr_forward() Method

Let's take a look at the ip_mr_forward() method:

```
static int ip_mr_forward(struct net *net, struct mr_table *mrt,
             struct sk_buff *skb, struct mfc_cache *cache,
             int local)
{
    int psend = -1;
    int vif, ct;
    int true_vifi = ipmr_find_vif(mrt, skb->dev);

    vif = cache->mfc_parent;
```

Here you can see update statistics of the resolved cache object (`mfc_un.res`):

```
cache->mfc_un.res.pkt++;
cache->mfc_un.res.bytes += skb->len;

if (cache->mfc_origin == htonl(INADDR_ANY) && true_vifi >= 0) {
    struct mfc_cache *cache_proxy;
```

The expression (`*, G`) means traffic from any source sending to the group G:

```
    /* For an (*,G) entry, we only check that the incomming
    * interface is part of the static tree.
    */
    cache_proxy = ipmr_cache_find_any_parent(mrt, vif);
    if (cache_proxy &&
        cache_proxy->mfc_un.res.ttls[true_vifi] < 255)
        goto forward;
}
/*
 * Wrong interface: drop packet and (maybe) send PIM assert.
 */
if (mrt->vif_table[vif].dev != skb->dev) {
    if (rt_is_output_route(skb_rtable(skb))) {
        /* It is our own packet, looped back.
         * Very complicated situation...
         *
         * The best workaround until routing daemons will be
         * fixed is not to redistribute packet, if it was
         * send through wrong interface. It means, that
         * multicast applications WILL NOT work for
         * (S,G), which have default multicast route pointing
         * to wrong oif. In any case, it is not a good
         * idea to use multicasting applications on router.
         */
        goto dont_forward;
    }

    cache->mfc_un.res.wrong_if++;

    if (true_vifi >= 0 && mrt->mroute_do_assert &&
        /* pimsm uses asserts, when switching from RPT to SPT,
         * so that we cannot check that packet arrived on an oif.
         * It is bad, but otherwise we would need to move pretty
         * large chunk of pimd to kernel. Ough... --ANK
         */
        (mrt->mroute_do_pim ||
        cache->mfc_un.res.ttls[true_vifi] < 255) &&
        time_after(jiffies,
               cache->mfc_un.res.last_assert + MFC_ASSERT_THRESH)) {
        cache->mfc_un.res.last_assert = jiffies;
```

Call the ipmr_cache_report() method to build an IGMP message (IGMPMSG_WRONGVIF) and to deliver it to the userspace multicast routing daemon by calling the sock_queue_rcv_skb() method:

```
        ipmr_cache_report(mrt, skb, true_vifi, IGMPMSG_WRONGVIF);
    }
    goto dont_forward;
}
```

The frame is now ready to be forwarded:

```
forward:
    mrt->vif_table[vif].pkt_in++;
    mrt->vif_table[vif].bytes_in += skb->len;

    /*
     *    Forward the frame
     */
    if (cache->mfc_origin == htonl(INADDR_ANY) &&
        cache->mfc_mcastgrp == htonl(INADDR_ANY)) {
        if (true_vifi >= 0 &&
            true_vifi != cache->mfc_parent &&
            ip_hdr(skb)->ttl >
                cache->mfc_un.res.ttls[cache->mfc_parent]) {
            /* It's an (*,*) entry and the packet is not coming from
             * the upstream: forward the packet to the upstream
             * only.
             */
            psend = cache->mfc_parent;
            goto last_forward;
        }
        goto dont_forward;
    }
    for (ct = cache->mfc_un.res.maxvif - 1;
         ct >= cache->mfc_un.res.minvif; ct--) {
        /* For (*,G) entry, don't forward to the incoming interface */
        if ((cache->mfc_origin != htonl(INADDR_ANY) ||
             ct != true_vifi) &&
            ip_hdr(skb)->ttl > cache->mfc_un.res.ttls[ct]) {
            if (psend != -1) {
                struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);
```

Call the ipmr_queue_xmit() method to continue with the packet forwarding:

```
                if (skb2)
                    ipmr_queue_xmit(net, mrt, skb2, cache,
                            psend);
            }
            psend = ct;
        }
    }
last_forward:
    if (psend != -1) {
        if (local) {
            struct sk_buff *skb2 = skb_clone(skb, GFP_ATOMIC);

            if (skb2)
                ipmr_queue_xmit(net, mrt, skb2, cache, psend);
        } else {
            ipmr_queue_xmit(net, mrt, skb, cache, psend);
            return 0;
        }
    }

dont_forward:
    if (!local)
        kfree_skb(skb);
    return 0;
}
```

Now that I have covered the multicast routing forwarding method, `ip_mr_forward()`, it is time to examine the `ipmr_queue_xmit()` method.

The ipmr_queue_xmit() Method

Let's take a look at the `ipmr_queue_xmit()` method:

```
static void ipmr_queue_xmit(struct net *net, struct mr_table *mrt,
                            struct sk_buff *skb, struct mfc_cache *c, int vifi)
{
        const struct iphdr *iph = ip_hdr(skb);
        struct vif_device *vif = &mrt->vif_table[vifi];
        struct net_device *dev;
        struct rtable *rt;
        struct flowi4 fl4;
```

The encap field is used when working with a tunnel:

```
        int encap = 0;

        if (vif->dev == NULL)
                goto out_free;

#ifdef CONFIG_IP_PIMSM
        if (vif->flags & VIFF_REGISTER) {
                vif->pkt_out++;
                vif->bytes_out += skb->len;
                vif->dev->stats.tx_bytes += skb->len;
                vif->dev->stats.tx_packets++;
                ipmr_cache_report(mrt, skb, vifi, IGMPMSG_WHOLEPKT);
                goto out_free;
        }
#endif
```

When working with a tunnel, a routing lookup is performed with the `vif->remote` and `vif->local`, which represent the destination and local addresses, respectively. These addresses are the end points of the tunnel. When working with a `vif_device` object which represents a physical device, a routing lookup is performed with the destination of the IPv4 header and 0 as a source address:

```
if (vif->flags & VIFF_TUNNEL) {
        rt = ip_route_output_ports(net, &fl4, NULL,
                                   vif->remote, vif->local,
                                   0, 0,
                                   IPPROTO_IPIP,
                                   RT_TOS(iph->tos), vif->link);
        if (IS_ERR(rt))
                goto out_free;
        encap = sizeof(struct iphdr);
} else {
       rt = ip_route_output_ports(net, &fl4, NULL, iph->daddr, 0,
                                  0, 0,
                                  IPPROTO_IPIP,
                                  RT_TOS(iph->tos), vif->link);
       if (IS_ERR(rt))
               goto out_free;
}

dev = rt->dst.dev;
```

Note that if the packet size is higher than the MTU, an ICMPv4 message is not sent (as is done in such a case under unicast forwarding); only the statistics are updated, and the packet is discarded:

```
if (skb->len+encap > dst_mtu(&rt->dst) && (ntohs(iph->frag_off) & IP_DF)) {
        /* Do not fragment multicasts. Alas, IPv4 does not
         * allow to send ICMP, so that packets will disappear
         * to blackhole.
         */

        IP_INC_STATS_BH(dev_net(dev), IPSTATS_MIB_FRAGFAILS);
        ip_rt_put(rt);
        goto out_free;
}

encap += LL_RESERVED_SPACE(dev) + rt->dst.header_len;

if (skb_cow(skb, encap)) {
        ip_rt_put(rt);
        goto out_free;
}

vif->pkt_out++;
vif->bytes_out += skb->len;

skb_dst_drop(skb);
skb_dst_set(skb, &rt->dst);
```

The TTL is decreased, and the IPv4 header checksum is recalculated (because the TTL is one of the IPv4 fields) when forwarding the packet; the same is done in the `ip_forward()` method for unicast packets:

```
ip_decrease_ttl(ip_hdr(skb));

/* FIXME: forward and output firewalls used to be called here.
 * What do we do with netfilter? -- RR
 */
if (vif->flags & VIFF_TUNNEL) {
        ip_encap(skb, vif->local, vif->remote);
        /* FIXME: extra output firewall step used to be here. --RR */
        vif->dev->stats.tx_packets++;
        vif->dev->stats.tx_bytes += skb->len;
}

IPCB(skb)->flags |= IPSKB_FORWARDED;

/*
 * RFC1584 teaches, that DVMRP/PIM router must deliver packets locally
 * not only before forwarding, but after forwarding on all output
 * interfaces. It is clear, if mrouter runs a multicasting
 * program, it should receive packets not depending to what interface
 * program is joined.
 * If we will not make it, the program will have to join on all
 * interfaces. On the other hand, multihoming host (or router, but
 * not mrouter) cannot join to more than one interface - it will
 * result in receiving multiple packets.
 */
```

Invoke the NF_INET_FORWARD hook:

```
        NF_HOOK(NFPROTO_IPV4, NF_INET_FORWARD, skb, skb->dev, dev,
                ipmr_forward_finish);
        return;

out_free:
        kfree_skb(skb);
}
```

The ipmr_forward_finish() Method

Let's take a look at the `ipmr_forward_finish()` method, which is a very short method—it is in fact identical to the `ip_forward()` method:

```
static inline int ipmr_forward_finish(struct sk_buff *skb)
{
        struct ip_options *opt = &(IPCB(skb)->opt);

        IP_INC_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTFORWDATAGRAMS);
        IP_ADD_STATS_BH(dev_net(skb_dst(skb)->dev), IPSTATS_MIB_OUTOCTETS, skb->len);
```

Handle IPv4 options, if set (see Chapter 4):

```
        if (unlikely(opt->optlen))
                ip_forward_options(skb);

        return dst_output(skb);
}
```

Eventually, `dst_output()` sends the packet via the `ip_mc_output()` method, which calls the `ip_finish_output()` method (both methods are in `net/ipv4/route.c`).

Now that I have covered these multicast methods, let's get a better understanding of how the value of the TTL field is used in multicast traffic.

The TTL in Multicast Traffic

The TTL field of the IPv4 header has a double meaning when discussing multicast traffic. The first is the same as in unicast IPV4 traffic: the TTL represents a hop counter which is decreased by 1 on every device that is forwarding the packet. When it reaches 0, the packet is discarded. This is done to avoid endless travelling of packets due to some error. The second meaning of the TTL, which is unique to multicast traffic, is a threshold. The TTL values are divided into scopes. Routers have a TTL threshold assigned to each of their interfaces, and only packets with a TTL greater than the interface's threshold are forwarded. Here are the values of these thresholds:

- *0:* Restricted to the same host (cannot be sent out by any interface)
- *1:* Restricted to the same subnet (will not be forwarded by a router)
- *32:* Restricted to the same site
- *64:* Restricted to the same region
- *128:* Restricted to the same continent
- *255:* Unrestricted in scope (global)

See: "IP Multicast Extensions for 4.3BSD UNIX and related systems," by Steve Deering, available at `www.kohala.com/start/mcast.api.txt`.

■ **Note** IPv4 Multicast Routing is implemented in `net/ipv4/ipmr.c`, `include/linux/mroute.h`, and `include/uapi/linux/mroute.h`.

This completes my discussion of Multicast Routing. The chapter now moves on to Policy Routing, which enables you to configure routing policies that are not based solely on the destination address.

Policy Routing

With Policy Routing, a system administrator can define up to 255 routing tables. This section discusses IPv4 Policy Routing; IPv6 Policy Routing is discussed in Chapter 8. In this section, I use the terms *policy* or *rule* for entries that are created by Policy Routing, in order to avoid confusing the ordinary routing entries (discussed in Chapter 5) with policy rules.

Policy Routing Management

Policy Routing management is done with the `ip rule` command of the `iproute2` package (there is no parallel for Policy Routing management with the `route` command). Let's see how to add, delete, and dump all Policy Routing rules:

- You add a rule with the `ip rule add` command; for example: `ip rule add tos 0x04 table 252`. After this rule is inserted, every packet which has an IPv4 TOS field matching 0x04 will be handled according to the routing rules of table 252. You can add routing entries to this table by specifying the table number when adding a route; for example: `ip route add default via 192.168.2.10 table 252`. This command is handled in the kernel by the `fib_nl_newrule()` method, in `net/core/fib_rules.c`. The `tos` modifier in the `ip rule` command earlier is one of the available SELECTOR modifiers of the `ip rule` command; see `man 8 ip rule`, and also Table 6-1 in the "Quick Reference" section at the end of this chapter.
- You delete a rule with the `ip rule del` command; for example: `ip rule del tos 0x04 table 252`. This command is handled in the kernel by the `fib_nl_delrule()` method in `net/core/fib_rules.c`.
- You dump all the rules with the `ip rule list` command or the `ip rule show` command. Both these commands are handled in the kernel by the `fib_nl_dumprule()` method in `net/core/fib_rules.c`.

You now have a good idea about the basics of Policy Routing management, so let's examine the Linux implementation of Policy Routing.

Policy Routing Implementation

The core infrastructure of Policy Routing is the `fib_rules` module, `net/core/fib_rules.c`. It is used by three protocols of the kernel networking stack: IPv4 (including the multicast module, which has a multicast policy routing feature, as mentioned in the "Multicast Routing" section earlier in this chapter), IPv6, and DECnet. The IPv4 Policy Routing is implemented also in a file named `fib_rules.c`. Don't be confused by the identical name (`net/ipv4/fib_rules.c`). In IPv6, policy routing is implemented in `net/ipv6/fib6_rules.c`. The header file, `include/net/fib_rules.h`, contains the data structures and methods of the Policy Routing core. Here is the definition of the `fib4_rule` structure, which is the basis for IPv4 Policy Routing:

```
struct fib4_rule {
    struct fib_rule    common;
    u8            dst_len;
    u8            src_len;
    u8            tos;
    __be32            src;
    __be32            srcmask;
    __be32            dst;
    __be32            dstmask;
#ifdef CONFIG_IP_ROUTE_CLASSID
    u32            tclassid;
#endif
};

(net/ipv4/fib_rules.c)
```

Three policies are created by default at boot time, by calling the fib_default_rules_init() method: the local (RT_TABLE_LOCAL) table, the main (RT_TABLE_MAIN) table, and the default (RT_TABLE_DEFAULT) table. Lookup is done by the fib_lookup() method. Note that there are two different implementations of the fib_lookup() method in include/net/ip_fib.h. The first one, which is wrapped in the #ifndef CONFIG_IP_MULTIPLE_TABLES block, is for non-Policy Routing, and the second is for Policy Routing. When working with Policy Routing, the lookup is performed like this: if there were no changes to the initial policy routing rules (net->ipv4.fib_has_custom_rules is not set), that means the rule must be in one of the three initial routing tables. So, first a lookup is done in the local table, then in the main table, and then the default table. If there is no corresponding entry, a network unreachable (-ENETUNREACH) error is returned. If there was some change in the initial policy routing rules (net->ipv4.fib_has_custom_rules is set), the_fib_lookup() method is invoked, which is a heavier method, because it iterates over the list of rules and calls fib_rule_match() for each rule in order to decide whether it matches or not. See the implementation of the fib_rules_lookup() method in net/core/fib_rules.c. (The fib_rules_lookup() method is invoked from the __fib_lookup() method). I should mention here that the net->ipv4.fib_has_custom_rules variable is set to false in the initialization phase, by the fib4_rules_init() method, and to true in the fib4_rule_configure() method and the fib4_rule_delete() method. Note that CONFIG_IP_MULTIPLE_TABLES should be set for working with Policy Routing.

This concludes my Multicast Routing discussion. The next section talks about Multipath Routing, which is the ability to add more than one nexthop to a route.

Multipath Routing

Multipath Routing provides the ability to add more than one nexthop to a route. Defining two nexthop nodes can be done like this, for example: ip route add default scope global nexthop dev eth0 nexthop dev eth1. A system administrator can also assign weights for each nexthop—like this, for example: ip route add 192.168.1.10 nexthop via 192.168.2.1 weight 3 nexthop via 192.168.2.10 weight 5. The fib_info structure represents an IPv4 routing entry that can have more than one FIB nexthop. The fib_nhs member of the fib_info object represents the number of FIB nexthop objects; the fib_info object contains an array of FIB nexthop objects named fib_nh. So in this case, a single fib_info object is created, with an array of two FIB nexthop objects. The kernel keeps the weight of each next hop in the nh_weight field of the FIB nexthop object (fib_nh). If weight was not specified when adding a multipath route, it is set by default to 1, in the fib_create_info() method. The fib_select_multipath() method is called to determine the nexthop when working with Multipath Routing. This method is invoked from two places: from the __ip_route_output_key() method, in the Tx path, and from the ip_mkroute_input() method, in the Rx path. Note that when the output device is set in the flow, the fib_select_multipath() method is not invoked, because the output device is known:

```
struct rtable *__ip_route_output_key(struct net *net, struct flowi4 *fl4) {
. . .
#ifdef CONFIG_IP_ROUTE_MULTIPATH
    if (res.fi->fib_nhs > 1 && fl4->flowi4_oif == 0)
        fib_select_multipath(&res);
    else
#endif
. . .

}
```

In the Rx path there is no need for checking whether fl4->flowi4_oif is 0, because it is set to 0 in the beginning of this method. I won't delve into the details of the fib_select_multipath() method. I will only mention that there is an element of randomness in the method, using jiffies, for helping in creating a fair weighted route distribution, and that the weight of each next hop is taken in account. The FIB nexthop to use is assigned by setting the FIB nexthop selector (`nh_sel`) of the specified `fib_result` object. In contrast to Multicast Routing, which is handled by a dedicated module (`net/ipv4/ipmr.c`), the code of Multipath Routing appears scattered in the existing routing code, enclosed in `#ifdef` CONFIG_IP_ROUTE_MULTIPATH conditionals, and no separate module was added in the source code for supporting it. As mentioned in Chapter 5, there was support for IPv4 multipath routing cache, but it was removed in 2007 in kernel 2.6.23; in fact, it never did work very well, and never got out of the experimental state. Do not confuse the removal of the multipath routing cache with the removal of the routing cache; these are two different caches. The removal of the routing cache took place five years later, in kernel 3.6 (2012).

■ **NOTE** CONFIG_IP_ROUTE_MULTIPATH should be set for Multipath Routing Support.

Summary

This chapter covered advanced IPv4 routing topics, like Multicast Routing, the IGMP protocol, Policy Routing, and Multipath Routing. You learned about the fundamental structures of Multicast Routing, such as the multicast table (`mr_table`), the multicast forwarding cache (MFC), the Vif device, and more. You also learned what should be done to set a host to be a multicast router, and all about the use of the `ttl` field in Multicast Routing. Chapter 7 deals with the Linux neighbouring subsystem. The "Quick Reference" section that follows covers the top methods related to the topics discussed in this chapter, ordered by their context.

Quick Reference

I conclude this chapter with a short list of important routing subsystem methods (some of which were mentioned in this chapter), a list of macros, and `procfs` multicast entries and tables.

Methods

Let's start with the methods:

int ip_mroute_setsockopt(struct sock *sk, int optname, char __user *optval, unsigned int optlen);

This method handles `setsockopt()` calls from the multicast routing daemon. The supported socket options are: MRT_INIT, MRT_DONE, MRT_ADD_VIF, MRT_DEL_VIF, MRT_ADD_MFC, MRT_DEL_MFC, MRT_ADD_MFC_PROXY, MRT_DEL_MFC_PROXY, MRT_ASSERT, MRT_PIM (when PIM support is set), and MRT_TABLE (when Multicast Policy Routing is set).

int ip_mroute_getsockopt(struct sock *sk, int optname, char __user *optval, int __user *optlen);

This method handles `getsockopt()` calls from the multicast routing daemon. The supported socket options are MRT_VERSION, MRT_ASSERT and MRT_PIM.

struct mr_table *ipmr_new_table(struct net *net, u32 id);

This method creates a new multicast routing table. The `id` of the table will be the specified `id.`

void ipmr_free_table(struct mr_table *mrt);

This method frees the specified multicast routing table and the resources attached to it.

int ip_mc_join_group(struct sock *sk , struct ip_mreqn *imr);

This method is for joining a multicast group. The address of the multicast group to be joined is specified in the given `ip_mreqn` object. The method returns 0 on success.

static struct mfc_cache *ipmr_cache_find(struct mr_table *mrt, __be32 origin, __be32 mcastgrp);

This method performs a lookup in the IPv4 multicast routing cache. It returns NULL when no entry is found.

bool ipv4_is_multicast(__be32 addr);

This method returns `true` if the address is a multicast address.

int ip_mr_input(struct sk_buff *skb);

This method is the main IPv4 multicast Rx method (`net/ipv4/ipmr.c`).

struct mfc_cache *ipmr_cache_alloc(void);

This method allocates a multicast forwarding cache (`mfc_cache`) entry.

static struct mfc_cache *ipmr_cache_alloc_unres(void);

This method allocates a multicast routing cache (`mfc_cache`) entry for the unresolved cache and sets the `expires` field of the queue of unresolved entries.

void fib_select_multipath(struct fib_result *res);

This method is called to determine the nexthop when working with Multipath Routing.

int dev_set_allmulti(struct net_device *dev, int inc);

This method increments/decrements the `allmulti` counter of the specified network device according to the specified increment (the increment can be a positive number or a negative number).

int igmp_rcv(struct sk_buff *skb);

This method is the receive handler for IGMP packets.

static int ipmr_mfc_add(struct net *net, struct mr_table *mrt, struct mfcctl *mfc, int mrtsock, int parent);

This method adds a multicast cache entry; it is invoked by calling `setsockopt()` from userspace with MRT_ADD_MFC.

static int ipmr_mfc_delete(struct mr_table *mrt, struct mfcctl *mfc, int parent);

This method deletes a multicast cache entry; it is invoked by calling `setsockopt()` from userspace with MRT_DEL_MFC.

static int vif_add(struct net *net, struct mr_table *mrt, struct vifctl *vifc, int mrtsock);

This method adds a multicast virtual interface; it is invoked by calling `setsockopt()` from userspace with MRT_ADD_VIF.

static int vif_delete(struct mr_table *mrt, int vifi, int notify, struct list_head *head);

This method deletes a multicast virtual interface; it is invoked by calling `setsockopt()` from userspace with MRT_DEL_VIF.

static void ipmr_expire_process(unsigned long arg);

This method removes expired entries from the queue of unresolved entries.

static int ipmr_cache_report(struct mr_table *mrt, struct sk_buff *pkt, vifi_t vifi, int assert);

This method builds an IGMP packet, setting the type in the IGMP header to be the specified `assert` value and the code to be `0`. This IGMP packet is delivered to the userspace multicast routing daemon by calling the `sock_queue_rcv_skb()` method. The `assert` parameter can be assigned one of these values: IGMPMSG_NOCACHE, when an unresolved cache entry is added to the queue of unresolved entries and wants to notify the userspace routing daemon that it should resolve it, IGMPMSG_WRONGVIF, and IGMPMSG_WHOLEPKT.

static int ipmr_device_event(struct notifier_block *this, unsigned long event, void *ptr);

This method is a notifier callback which is registered by the `register_netdevice_notifier()` method; when some network device is unregistered, a NETDEV_UNREGISTER event is generated; this callback receives this event and deletes the `vif_device` objects in the `vif_table`, whose device is the one that was unregistered.

static void mrtsock_destruct(struct sock *sk);

This method is called when the userspace routing daemon calls `setsockopt()` with MRT_DONE. This method nullifies the multicast routing socket (`mroute_sk` of the multicast routing table), decrements the `mc_forwarding` `procfs` entry, and calls the `mroute_clean_tables()` method to free resources.

Macros

This section describes our macros.

MFC_HASH(a,b)

This macro calculates the hash value for adding entries to the MFC cache. It takes the group multicast address and the source IPv4 address as parameters.

VIF_EXISTS(_mrt, _idx)

This macro checks the existence of an entry in the `vif_table`; it returns `true` if the array of multicast virtual devices (`vif_table`) of the specified multicast routing table (`mrt`) has an entry with the specified index (`_idx`).

Procfs Multicast Entries

The following is a description of two important `procfs` multicast entries:

/proc/net/ip_mr_vif

Lists all the multicast virtual interfaces; it displays all the `vif_device` objects in the multicast virtual device table (`vif_table`). Displaying the `/proc/net/ip_mr_vif entry` is handled by the `ipmr_vif_seq_show()` method.

/proc/net/ip_mr_cache

The state of the Multicast Forwarding Cache (MFC). This entry shows the following fields of all the cache entries: group multicast address (`mfc_mcastgrp`), source IP address (`mfc_origin`), input interface index (`mfc_parent`), forwarded packets (`mfc_un.res.pkt`), forwarded bytes (`mfc_un.res.bytes`), wrong interface index (`mfc_un.res.wrong_if`), the index of the forwarding interface (an index in the `vif_table`), and the entry in the `mfc_un.res.ttls` array corresponding to this index. Displaying the `/proc/net/ip_mr_cache` entry is handled by the `ipmr_mfc_seq_show()` method.

Table

And finally, here in Table 6-1, is the table of rule selectors.

Table 6-1. *IP Rule Selectors*

Linux Symbol	Selector	Member of fib_rule	fib4_rule
FRA_SRC	from	src	(fib4_rule)
FRA_DST	to	dst	(fib4_rule)
FRA_IIFNAME	iif	iifname	(fib_rule)
FRA_OIFNAME	oif	oifname	(fib_rule)
FRA_FWMARK	fwmark	mark	(fib_rule)
FRA_FWMASK	fwmark/fwmask	mark_mask	(fib_rule)
FRA_PRIORITY	preference,order,priority	pref	(fib_rule)
-	tos, dsfield	tos	(fib4_rule)